Université Hassan 1er
Faculté des Sciences et Techniques - Settat

TP Sécurité : Manipulation de la Cryptographie en Python

MD5 – DES – AES – PBKDF2 – RSA – Fernet – Hashlib – Secrets

Réalisé par :
Abdeljalil BOUZINE

Encadrant :
HAMID GARMANI

Date : Décembre 2025

Table des matières

Introduction

Ce rapport présente l'ensemble des manipulations cryptographiques réalisées en Python dans le cadre du TP de sécurité informatique. L'objectif est de comprendre et expérimenter plusieurs primitives cryptographiques fondamentales : hachage, chiffrement symétrique, dérivation de clé, chiffrement asymétrique, signature numérique, génération sécurisée et vérification d'intégrité.

Les bibliothèques utilisées sont :

- **PyCryptodome** : DES, AES, PBKDF2, RSA
- **cryptography** : Fernet, PBKDF2HMAC, AES bas niveau
- **hashlib** : SHA-256, SHA-512
- **secrets** : mots de passe et tokens sécurisés

Chaque section contient : explications, code Python exécuté, zones pour résultats et captures.

Chapitre 1

Manipulations avec PyCryptodome

1.1 Hachage MD5 d'un fichier

Objectif : générer l'empreinte MD5 d'un fichier et vérifier que le moindre changement modifie l'empreinte.

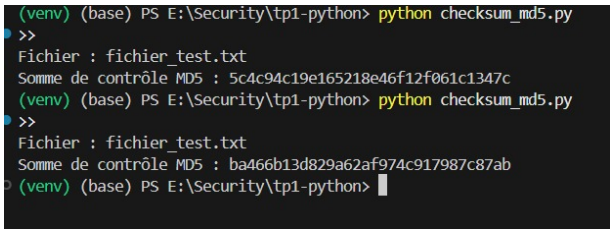
Code Python

```
from Crypto.Hash import MD5

def get_file_checksum(filename: str) -> str:
    h = MD5.new()
    chunk_size = 8192

    with open(filename, "rb") as f:
        while True:
            chunk = f.read(chunk_size)
            if not chunk:
                break
            h.update(chunk)
    return h.hexdigest()

if __name__ == "__main__":
    nom_fichier = "fichier_test.txt"
    print("Fichier␣:", nom_fichier)
    print("Somme␣de␣contr le␣MD5␣:", get_file_checksum(nom_fichier
        ))
```

Explication du code

— **Importation du module MD5 :** La bibliothèque `Crypto.Hash.MD5` fournit une implémentation de l'algorithme de hachage MD5, permettant de générer une empreinte unique à partir d'un contenu quelconque.
— **Création d'un objet de hachage :** `h = MD5.new()` initialise un nouvel objet MD5. Cet objet maintient l'état du calcul du hash au fur et à mesure que des données lui sont ajoutées.

— **Lecture du fichier par blocs :** Le fichier est ouvert en mode binaire (`"rb"`) afin de lire exactement son contenu brut. Il est lu morceau par morceau via des blocs de 8192 octets pour éviter de charger des fichiers trop volumineux en mémoire.
— **Mise à jour progressive du hash :** À chaque itération, `h.update(chunk)` ajoute les données du bloc au calcul MD5. Cela permet de calculer le hash sur des flux de données continus, même très grands.
— **Condition d'arrêt :** Lorsque `f.read()` renvoie un bloc vide, cela signifie que le fichier a été entièrement parcouru, et la boucle s'arrête.
— **Récupération du hash final :** La méthode `h.hexdigest()` retourne l'empreinte MD5 sous forme hexadécimale lisible. C'est cette valeur qu'on utilise pour vérifier l'intégrité d'un fichier.
— **Bloc principal :** Le fichier `fichier_test.txt` est haché, puis le hash est affiché. Si le fichier change, ne serait-ce qu'un seul caractère, l'empreinte MD5 devient complètement différente.
— **Usage :** Ce hash permet de détecter toute modification accidentelle ou intentionnelle d'un fichier, mais MD5 n'est plus considéré comme sécurisé pour des usages cryptographiques avancés.



1.2 Chiffrement et déchiffrement DES

```python
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad

user_text = "user"
message_text = "message"
key = b"mycipher"

user = pad(user_text.encode(), DES.block_size)
message = pad(message_text.encode(), DES.block_size)

cipher = DES.new(key, DES.MODE_ECB)
cipher_user = cipher.encrypt(user)
cipher_message = cipher.encrypt(message)

decipher = DES.new(key, DES.MODE_ECB)
dec_user = unpad(decipher.decrypt(cipher_user), DES.block_size)
dec_message = unpad(decipher.decrypt(cipher_message), DES.
    block_size)

print(dec_user, dec_message)
```

Explication du code

- **Importation des modules :** Le module `DES` permet de créer un chiffreur utilisant l'algorithme DES. Le module `pad/unpad` sert à adapter la taille des données au bloc DES.
- **Préparation des textes :** Les chaînes `"user"` et `"message"` sont encodées en bytes puis complétées (*padding*) pour atteindre une taille multiple de 8 octets, car DES chiffre uniquement des blocs de 64 bits.
- **Clé DES :** La clé `b"mycipher"` contient 8 octets, ce qui correspond exactement à la taille d'une clé DES (64 bits). Une clé de toute autre longueur rendrait l'algorithme invalide.
- **Création de l'objet de chiffrement :** `DES.new(key, DES.MODE_ECB)` crée un chiffreur en mode ECB (*Electronic Code Book*). Ce mode chiffre chaque bloc indépendamment, ce qui est simple mais peu sécurisé.
- **Chiffrement :** Les fonctions `cipher.encrypt()` produisent un résultat binaire chiffré pour `user` et `message`. Chaque sortie dépend de la clé et du texte d'entrée.
- **Déchiffrement :** Un second objet `DES.new()` est créé pour le déchiffrement (obligatoire en PyCryptodome). Les données chiffrées sont déchiffrées puis `unpad()` retire le remplissage ajouté avant le chiffrement.
- **Affichage final :** Les textes obtenus après déchiffrement sont identiques aux textes initiaux, ce qui confirme le bon fonctionnement du chiffrement DES.

```
(venv) (base) PS E:\Security\tp1-python> python des_encrypt_decrypt.py
>>
User chiffré   : b'\xaeK\xa2\x87/\xc1!\xb3'
Message chiffré: b'V\xe2\x04WT\x98\x87\x80'
User déchiffré   : user
Message déchiffré: message
(venv) (base) PS E:\Security\tp1-python>
```

1.3 AES CBC simple

```python
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import random, string

key_str = ''.join(random.choice(string.ascii_letters + string.
    digits) for _ in range(16))
key = key_str.encode()
iv = b"This is an IV-12"

encryptor = AES.new(key, AES.MODE_CBC, iv)
decryptor = AES.new(key, AES.MODE_CBC, iv)

def aes_encrypt(plaintext):
    return encryptor.encrypt(pad(plaintext, AES.block_size))

def aes_decrypt(ciphertext):
    return unpad(decryptor.decrypt(ciphertext), AES.block_size)

plaintext = b"This is the secret message "
```

```
ciphertext = aes_encrypt(plaintext)
decrypted = aes_decrypt(ciphertext)
```

Explication du code

- **Importation des modules :** Les fonctions de chiffrement AES proviennent de `Crypto.Cipher`, tandis que `pad` et `unpad` permettent de gérer le remplissage nécessaire pour que les données aient une taille compatible avec le chiffrement en mode bloc.
- **Génération d'une clé AES :** Une chaîne de 16 caractères aléatoires est créée, puis convertie en bytes pour servir de clé AES. Une clé AES de 16 octets correspond à AES-128 bits.
- **Initialisation Vector (IV) :** L'IV est une suite de 16 octets utilisée pour le mode CBC (Cipher Block Chaining). Il garantit que deux messages identiques ne produisent jamais le même chiffrement, évitant les répétitions dans les blocs chiffrés. Ici, l'IV est fixe pour simplifier le TP, mais en pratique il doit toujours être aléatoire.
- **Création des objets de chiffrement et déchiffrement :** `AES.new(key, AES.MODE_CBC, iv)` instancie le chiffreur AES en mode CBC. Le même IV et la même clé doivent être utilisés pour chiffrer et déchiffrer.
- **Fonction de chiffrement** `aes_encrypt()` :
  - Les données sont d'abord paddées pour atteindre un multiple de 16 octets, taille obligatoire d'un bloc AES.
  - Le chiffreur produit un texte chiffré binaire incompréhensible sans la clé.
- **Fonction de déchiffrement** `aes_decrypt()` :
  - Les données chiffrées sont déchiffrées bloc par bloc.
  - Le *padding* ajouté avant le chiffrement est retiré avec `unpad` pour retrouver le texte original.
- **Exécution finale :** Le message `"This is the secret message"` est chiffré puis déchiffré. Le résultat obtenu après déchiffrement doit être identique au texte initial, ce qui confirme la validité de la procédure AES CBC.

1.4 AES sur fichier

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad

def encrypt_file(key, input_file, output_file):
    iv = get_random_bytes(16)
    cipher = AES.new(key, AES.MODE_CBC, iv)

    data = open(input_file, "rb").read()
```

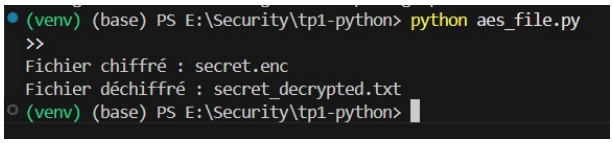
```
    ciphertext = cipher.encrypt(pad(data, AES.block_size))

    open(output_file, "wb").write(iv + ciphertext)

def decrypt_file(key, input_file, output_file):
    file_data = open(input_file, "rb").read()
    iv, ciphertext = file_data[:16], file_data[16:]

    cipher = AES.new(key, AES.MODE_CBC, iv)
    plaintext = unpad(cipher.decrypt(ciphertext), AES.block_size)

    open(output_file, "wb").write(plaintext)
```

Explication du code

- **Importation des modules :**
  - `AES` pour le chiffrement symétrique,
  - `get_random_bytes` pour générer un IV aléatoire,
  - `pad/unpad` pour ajuster la taille des données.
- **Fonction `encrypt_file()` :** Cette fonction chiffre tout le contenu d'un fichier.
  - Un **IV aléatoire de 16 octets** est généré, ce qui est indispensable en mode CBC pour garantir la sécurité.
  - L'objet `cipher` est créé avec la clé AES et l'IV.
  - Le fichier d'entrée est lu en mode binaire (`rb`) pour récupérer son contenu brut.
  - Les données sont paddées pour atteindre une longueur multiple de 16 octets, taille nécessaire pour AES.
  - Le texte chiffré est écrit dans le fichier de sortie sous la forme :

    $$IV \,||\, donneschiffres$$

    Ce format permet de récupérer facilement l'IV lors du déchiffrement.
- **Fonction `decrypt_file()` :** Cette fonction effectue l'opération inverse.
  - Le fichier chiffré est lu intégralement en mémoire.
  - Les **16 premiers octets** correspondent à l'IV, le reste correspond au texte chiffré.
  - Un nouvel objet `cipher` est instancié avec la même clé et le même IV qu'au chiffrement.
  - Le texte chiffré est déchiffré bloc par bloc.
  - Le padding ajouté avant le chiffrement est retiré avec `unpad()` pour retrouver les données originales.
  - Le fichier déchiffré est écrit en clair dans le fichier de sortie.
- **Structure des fichiers chiffrés :** Le choix d'écrire **IV + ciphertext** dans un seul fichier est une pratique standard en cryptographie. L'IV n'est pas secret : seule la clé doit rester confidentielle.
- **Conclusion :** Ce programme permet de chiffrer et déchiffrer n'importe quel fichier (texte, PDF, image...) en garantissant la confidentialité grâce à AES-CBC.



1.5 PBKDF2 + AES CBC

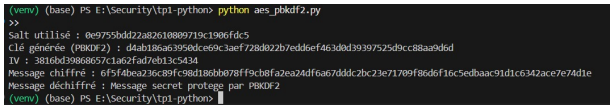
```python
from Crypto.Protocol.KDF import PBKDF2
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad

password = "mypassword123"
salt = get_random_bytes(16)
key = PBKDF2(password, salt, dkLen=32)

iv = get_random_bytes(16)
cipher = AES.new(key, AES.MODE_CBC, iv)

plaintext = b"Message␣secret␣protege␣par␣PBKDF2"
ciphertext = cipher.encrypt(pad(plaintext, AES.block_size))

decipher = AES.new(key, AES.MODE_CBC, iv)
decrypted = unpad(decipher.decrypt(ciphertext), AES.block_size)
```

Explication du code

- **Objectif général :** Ce code combine deux mécanismes cryptographiques importants :
  - la dérivation de clé sécurisée **PBKDF2**,
  - le chiffrement symétrique **AES en mode CBC**.

  L'objectif est de transformer un mot de passe humain en une clé robuste, puis de l'utiliser pour chiffrer un message.
- **Mot de passe et sel (salt) :**
  - Le mot de passe `"mypassword123"` est volontairement simple pour le TP.
  - `get_random_bytes(16)` génère un **sel aléatoire de 16 octets**.
  - Le sel empêche l'utilisation de tables arc-en-ciel et garantit que deux utilisateurs ayant le même mot de passe obtiennent des clés différentes.
- **Dérivation de clé PBKDF2 :** `PBKDF2(password, salt, dkLen=32)` produit une clé de 32 octets, soit une clé **AES-256 bits**. PBKDF2 applique de nombreuses itérations de hachage interne pour rendre l'extraction de la clé très coûteuse pour un attaquant.
- **Génération de l'IV :** `get_random_bytes(16)` génère un vecteur d'initialisation unique pour AES-CBC. Comme toujours, l'IV n'a pas besoin d'être secret, mais doit être imprévisible.
- **Chiffrement du message :**
  - Les données sont paddées pour respecter la taille d'un bloc AES (16 octets).

  - `cipher.encrypt()` produit le texte chiffré (*ciphertext*), illisible sans la clé dérivée.
- **Déchiffrement :**
  - Un nouvel objet `AES.new()` est nécessaire car chaque objet ne peut servir qu'une seule fois.
  - Le message est déchiffré bloc par bloc.
  - `unpad()` retire le remplissage pour retrouver le message original.
- **Importance de cette approche :** Cette méthode est beaucoup plus sécurisée que d'utiliser directement un mot de passe comme clé AES, car :
  - PBKDF2 rend les attaques par force brute très coûteuses,
  - le sel empêche les attaques par dictionnaire pré-calculé,
  - AES-256 garantit un chiffrement robuste.
- **Conclusion :** Ce code montre une méthode standard et recommandée pour protéger des données à partir d'un mot de passe utilisateur.



1.6 RSA : génération, signature, vérification, chiffrement

```python
from Crypto.PublicKey import RSA
from Crypto.Signature import pkcs1_15
from Crypto.Hash import SHA256
from Crypto.Cipher import PKCS1_OAEP

def generate_keys():
    key = RSA.generate(2048)
    open("private.pem","wb").write(key.export_key())
    open("public.pem","wb").write(key.publickey().export_key())

def sign_message():
    message = b"Voici un message important"
    h = SHA256.new(message)
    private_key = RSA.import_key(open("private.pem","rb").read())
    signature = pkcs1_15.new(private_key).sign(h)
    open("signature.bin","wb").write(signature)
    return message

def verify_signature(message):
    public_key = RSA.import_key(open("public.pem","rb").read())
    signature = open("signature.bin","rb").read()
    h = SHA256.new(message)
    pkcs1_15.new(public_key).verify(h, signature)

def rsa_encrypt():
    public_key = RSA.import_key(open("public.pem","rb").read())
```

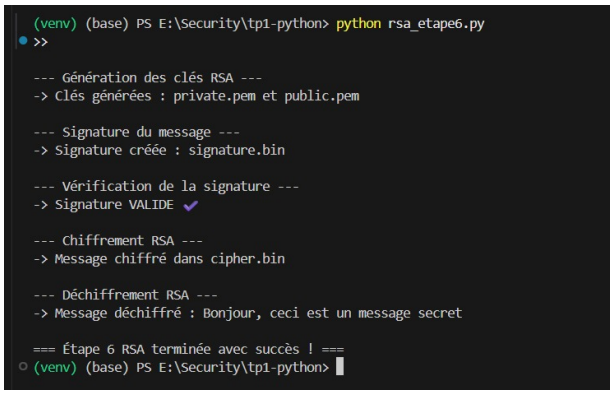
```
27      cipher = PKCS1_OAEP.new(public_key)
28      ciphertext = cipher.encrypt(b"Bonjour , ceci est un message
            secret")
29      open("cipher.bin","wb").write(ciphertext)
30
31  def rsa_decrypt():
32      private_key = RSA.import_key(open("private.pem","rb").read())
33      ciphertext = open("cipher.bin","rb").read()
34      plaintext = PKCS1_OAEP.new(private_key).decrypt(ciphertext)
35      print(plaintext)
```

Explication du code

- **Objectif général :** Ce programme illustre les quatre opérations principales de la cryptographie RSA :
  1. la génération d'une paire de clés RSA (publique/privée),
  2. la signature numérique d'un message,
  3. la vérification de signature,
  4. le chiffrement et déchiffrement RSA avec OAEP.
- **Génération des clés RSA :**
  - `RSA.generate(2048)` crée une clé privée RSA de 2048 bits (taille recommandée).
  - La clé privée est enregistrée dans `private.pem`.
  - La clé publique correspondante est enregistrée dans `public.pem`.
  - Les fichiers PEM suivent un format standard utilisé en cryptographie.
- **Signature d'un message :**
  - Le message est converti en hash SHA-256 : c'est ce hash qui est signé et non le message complet.
  - La clé privée est chargée depuis `private.pem`.
  - `pkcs1_15.new(private_key).sign(h)` applique l'algorithme standard PKCS#1 v1.5.
  - La signature produite est enregistrée dans `signature.bin`.
  - Le retour de la fonction permet de vérifier ensuite la signature sur le même message.
- **Vérification de la signature :**
  - La clé publique est chargée depuis `public.pem`.
  - La signature est relue depuis `signature.bin`.
  - Le programme recalcule le hash du message.
  - Si la vérification échoue (modification du message ou mauvaise clé), une exception serait déclenchée.
  - Si aucune exception n'est levée, la signature est considérée comme **valide**.
- **Chiffrement RSA (OAEP) :**
  - L'algorithme OAEP est utilisé car il est plus sécurisé que le chiffrement RSA "brut".
  - Le chiffrement s'effectue avec la clé publique (*tout le monde peut chiffrer*).
  - Le message chiffré est sauvegardé dans `cipher.bin`.
- **Déchiffrement RSA (OAEP) :**

  - Seule la clé privée peut déchiffrer le message (*seul le destinataire peut lire*).
  - Le fichier chiffré `cipher.bin` est lu en mémoire.
  - Le plaintext est récupéré et affiché.
  - Cette opération confirme le bon fonctionnement du chiffrement RSA.
- **Conclusion :** Ce code illustre le principe fondamental de RSA :
  - **signature = clé privée**, vérifiable avec la clé publique,
  - **chiffrement = clé publique**, déchiffrable uniquement avec la clé privée.

  Il s'agit d'une base solide pour comprendre l'authentification numérique et l'échange de données sécurisé.



Chapitre 2

Manipulations avec le module `cryptography`

2.1 Fernet : chiffrement de texte

```python
from cryptography.fernet import Fernet

def generate_key():
    key = Fernet.generate_key()
    open("fernet.key","wb").write(key)
    return key

key = generate_key()
f = Fernet(key)

message = b"Message secret avec Fernet"
token = f.encrypt(message)
plaintext = f.decrypt(token)
```

Explication du code

- **Objectif général :** Le module `Fernet` de la bibliothèque `cryptography` permet d'effectuer un chiffrement symétrique moderne, sécurisé et facile à utiliser. Il garantit :
  - la **confidentialité** (données chiffrées),
  - l'**intégrité** (détection de modification),
  - l'**authenticité** (clé unique nécessaire pour déchiffrer).
- **Génération de la clé Fernet :**
  - `Fernet.generate_key()` crée une clé de 32 octets encodée en Base64.
  - Cette clé contient l'ensemble des paramètres nécessaires au chiffrement : AES-128 en CBC + HMAC-SHA256 pour vérifier l'intégrité.
  - La clé est enregistrée dans le fichier `fernet.key` pour être réutilisée.
- **Création de l'objet Fernet :** `f = Fernet(key)` instancie un objet chiffre/déchiffre sécurisé. Cet objet encapsule automatiquement toutes les opérations cryptographiques complexes.
- **Chiffrement du message :**
  - Le message en clair est défini en bytes : `b"Message secret avec Fernet"`.
  - L'appel `f.encrypt(message)` génère un **token** comprenant :

  - un timestamp,
  - l'IV,
  - le ciphertext,
  - la signature HMAC.
- Ce token est sûr à stocker ou transmettre : toute modification entraînerait une erreur au déchiffrement.

— **Déchiffrement :**
- `f.decrypt(token)` vérifie d'abord l'intégrité via HMAC.
- Ensuite, Fernet déchiffre automatiquement le message AES-CBC.
- Le résultat retourné est exactement le texte original.

— **Conclusion :** Fernet facilite fortement le chiffrement symétrique tout en assurant des propriétés cryptographiques avancées. C'est une solution idéale pour :
- chiffrer des messages sensibles,
- stocker des données de configuration sécurisées,
- protéger des informations dans des applications Python.

2.2 Fernet : chiffrement de fichier

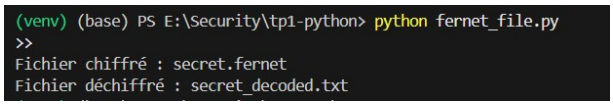
```python
from cryptography.fernet import Fernet

def load_key():
    return open("fernet.key","rb").read()

def encrypt_file(input_file, output_file, key):
    f = Fernet(key)
    data = open(input_file,"rb").read()
    encrypted = f.encrypt(data)
    open(output_file,"wb").write(encrypted)

def decrypt_file(input_file, output_file, key):
    f = Fernet(key)
    encrypted = open(input_file,"rb").read()
    decrypted = f.decrypt(encrypted)
    open(output_file,"wb").write(decrypted)

key = load_key()
encrypt_file("secret.txt","secret.fernet",key)
decrypt_file("secret.fernet","secret_decoded.txt",key)
```

Explication du code

— **Objectif général :** Cette section montre comment utiliser l'algorithme Fernet pour chiffrer et déchiffrer des fichiers complets (texte, PDF, image...). Fernet assure simultanément :

  - le chiffrement AES-CBC,
  - l'intégrité via un HMAC,
  - une structure sécurisée du token.

  Cela garantit que le fichier ne peut pas être modifié ou lu sans la clé Fernet.
- **Chargement de la clé Fernet :** La fonction `load_key()` relit simplement la clé stockée dans le fichier `fernet.key`. Cette clé doit être la même que celle utilisée pour chiffrer et déchiffrer, ce qui fait de Fernet un système de chiffrement symétrique.
- **Fonction `encrypt_file()` : chiffrement du fichier**
  - L'objet Fernet est initialisé avec la clé fournie.
  - Le fichier d'entrée (`input_file`) est lu en mode binaire pour récupérer son contenu exact.
  - `f.encrypt(data)` chiffre totalement le contenu du fichier et produit un token sécurisé.
  - Le fichier chiffré est écrit dans `output_file`.
  - Le format Fernet du résultat inclut automatiquement :
    - un timestamp,
    - un IV AES-CBC aléatoire,
    - le ciphertext,
    - une signature HMAC-SHA256.
- **Fonction `decrypt_file()` : déchiffrement du fichier**
  - Le fichier chiffré est relu intégralement.
  - La méthode `f.decrypt()` :
    - vérifie d'abord l'authenticité et l'intégrité du fichier (HMAC),
    - déchiffre ensuite le contenu AES-CBC.
  - Le contenu original du fichier est récupéré et sauvegardé dans le fichier de sortie.
- **Programme principal :**
  - La clé est chargée depuis le fichier `fernet.key`.
  - Le fichier `secret.txt` est chiffré dans `secret.fernet`.
  - Le fichier chiffré est ensuite déchiffré dans `secret_decoded.txt`.
  - Le fichier final est identique au fichier original, preuve que le processus fonctionne correctement.
- **Intérêt du chiffrement Fernet pour les fichiers :**
  - idéal pour protéger des données sensibles (mots de passe, configs, documents...),
  - détection automatique de modification : le fichier ne peut pas être altéré,
  - simplicité d'utilisation avec une sécurité robuste sous-jacente.



2.3 PBKDF2HMAC + Fernet

```
1 import base64, os
2 from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
3 from cryptography.hazmat.primitives import hashes
4 from cryptography.fernet import Fernet
```

```
5
6  def derive_key_from_password(password, salt):
7      kdf = PBKDF2HMAC(
8          algorithm=hashes.SHA256(),
9          length=32,
10         salt=salt,
11         iterations=100000,
12     )
13     return base64.urlsafe_b64encode(kdf.derive(password.encode()))
14
15 salt = os.urandom(16)
16 key = derive_key_from_password("monSuperMotDePasse123", salt)
17 f = Fernet(key)
```

Explication du code

- **Objectif général :** Ce code montre comment dériver une **clé cryptographique sécurisée à partir d'un mot de passe** en utilisant PBKDF2-HMAC, puis comment l'utiliser avec Fernet pour chiffrer et déchiffrer des données. Cette technique transforme un mot de passe classique en une clé robuste, résistante aux attaques par force brute.
- **Importation des modules :**
  - `PBKDF2HMAC` : algorithme de dérivation de clé sécurisé.
  - `hashes.SHA256` : fonction de hachage interne utilisée par PBKDF2.
  - `Fernet` : module de chiffrement symétrique AES+HMAC.
  - `base64` : permet de convertir la clé en format compatible Fernet.
  - `os.urandom` : permet de générer un sel aléatoire sécurisé.
- **Fonction `derive_key_from_password()` :**
  - Le KDF PBKDF2 est initialisé avec :
    - l'algorithme de hachage SHA-256,
    - une longueur de sortie de 32 octets (clé AES-256),
    - un **sel** aléatoire de 16 octets,
    - **100 000 itérations**, rendant les attaques très coûteuses.
  - La fonction `derive()` applique PBKDF2 au mot de passe fourni.
  - La clé obtenue est encodée en Base64 URL-safe afin d'être compatible avec Fernet.
- **Rôle du sel (salt) :**
  - `os.urandom(16)` génère 16 octets aléatoires imprévisibles.
  - Le sel empêche l'utilisation de bases de données pré-calculées (rainbow tables).
  - Deux utilisateurs avec le même mot de passe obtiendront des clés **différentes**.
- **Génération de la clé finale :**
  - Le mot de passe `"monSuperMotDePasse123"` est transformé en clé cryptographique solide.
  - Cette clé peut ensuite servir au chiffrement symétrique sécurisé.
- **Création de l'objet Fernet :** `f = Fernet(key)` crée un chiffreur Fernet utilisant la clé dérivée. Cet objet permet ensuite de chiffrer/déchiffrer n'importe quelle donnée sensible.
- **Avantages de l'approche PBKDF2HMAC + Fernet :**

— transformation d'un mot de passe simple en clé AES-256 robuste,
— protection contre les attaques par dictionnaire,
— compatibilité directe avec les mécanismes sécurisés de Fernet (AES-CBC + HMAC),
— méthode recommandée pour stocker ou transmettre des données sensibles.

2.4 AES bas niveau (Cipher, CBC)

```python
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives.ciphers import Cipher,
    algorithms, modes
import os

def encrypt_aes(key, iv, data):
    padder = padding.PKCS7(128).padder()
    padded = padder.update(data) + padder.finalize()
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    return cipher.encryptor().update(padded)

def decrypt_aes(key, iv, ciphertext):
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    padded = cipher.decryptor().update(ciphertext)
    return padding.PKCS7(128).unpadder().update(padded)

key = os.urandom(32)
iv = os.urandom(16)
```

Explication du code

— **Objectif général :** Ce code montre comment effectuer un chiffrement AES manuellement en utilisant les primitives bas niveau du module `cryptography`. Contrairement à Fernet (haut niveau), ici toutes les étapes sont explicitement gérées : padding, mode CBC, génération des clés, chiffrement et déchiffrement.
— **Importation des modules :**
  — `padding.PKCS7` : ajoute/retire du remplissage pour obtenir une taille multiple de 16 octets.
  — `Cipher, algorithms, modes` : primitives cryptographiques bas niveau.
  — `os.urandom` : utilisé pour générer des clés et IV sécurisés.
— **Fonction `encrypt_aes()` : chiffrement AES**
  — Le padding PKCS#7 est appliqué pour ajuster la taille des données à un multiple de 128 bits (16 octets).
  — Un objet `Cipher` est créé avec :
    — l'algorithme AES,

  - la clé secrète,
  - le mode CBC et son IV.
- L'appel à `encryptor().update(padded)` chiffre le message en blocs consécutifs.
- La fonction retourne le **ciphertext** (texte chiffré).

— **Fonction `decrypt_aes()` : déchiffrement AES**
- Un nouvel objet `Cipher` est recréé (car chaque objet ne peut servir qu'une seule fois).
- Le ciphertext est déchiffré bloc par bloc.
- Le résultat contient encore le padding, qui est retiré avec :

$$PKCS7(128).unpadder()$$

- La fonction retourne le texte original.

— **Génération de la clé et de l'IV :**
- `key = os.urandom(32)` génère une clé AES-256 bits (32 octets).
- `iv = os.urandom(16)` produit un IV unique de 16 octets, indispensable pour le mode CBC.

— **Importance de cette approche bas niveau :** Elle permet :
- de comprendre le mécanisme interne d'AES,
- de gérer manuellement le padding, l'IV, les flux d'encryption,
- de personnaliser les schémas de chiffrement non fournis par Fernet.

— **Conclusion :** Ce code illustre un schéma AES-CBC "manuel" fidèle aux standards cryptographiques modernes. C'est une étape essentielle pour comprendre ce qui se passe "sous le capot" des bibliothèques plus haut niveau comme Fernet.

Chapitre 3

Génération sécurisée et hachage

3.1 Génération d'un mot de passe aléatoire sécurisé

```
1 import secrets, string
2
3 alphabet = string.ascii_letters + string.digits + string.
     punctuation
4 password = ''.join(secrets.choice(alphabet) for _ in range(16))
5 print("Mot de passe s curis :", password)
```

Explication du code

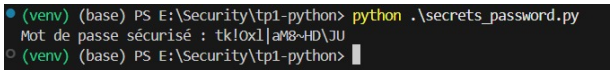
- **Objectif général :** Ce code permet de générer un mot de passe aléatoire et sécurisé en utilisant le module `secrets`, qui fournit un générateur cryptographiquement sûr, plus robuste que le module `random`.
- **Importation des modules :**
  - `secrets` : utilisé pour produire des valeurs aléatoires imprévisibles, adaptées à la cryptographie.
  - `string` : fournit les ensembles standard de caractères (lettres, chiffres, ponctuation).
- **Construction de l'alphabet :** `alphabet = string.ascii_letters + string.digits + string.punctuation` crée une chaîne qui contient :
  - toutes les lettres minuscules et majuscules (A–Z, a–z),
  - les chiffres (0–9),
  - les symboles et caractères spéciaux.

  Cela rend le mot de passe plus varié et plus difficile à deviner.
- **Génération du mot de passe sécurisé :** L'instruction :

  ```
  ''.join(secrets.choice(alphabet) for _ in range(16))
  ```

  sélectionne 16 caractères aléatoires dans l'alphabet. Grâce à `secrets.choice()`, chaque caractère :
  - est tiré à partir d'un générateur sécurisé,
  - est imprévisible même par un attaquant disposant d'outils avancés.
- **Affichage du mot de passe final :** Le mot de passe généré est affiché directement. Comme il contient lettres, chiffres et symboles, il respecte les bonnes pratiques en matière de robustesse.

— **Conclusion :** Ce code illustre une méthode standard pour générer des mots de passe résistants aux attaques :
  — force brute,
  — dictionnaires,
  — prédictions pseudo-aléatoires.
  L'utilisation du module `secrets` est recommandée pour toute application nécessitant de la sécurité (comptes utilisateurs, clés temporaires, tokens, etc.).

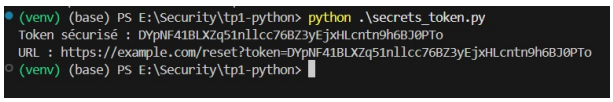


3.2 Générer un token sécurisé

```python
import secrets

token = secrets.token_urlsafe(32)
print("Token␣s curis ␣:", token)
```

Explication du code

— **Objectif :** Générer un token aléatoire, long et sécurisé, utilisable pour des opérations sensibles comme l'authentification, le reset de mot de passe, ou la création de liens temporaires.
— **Module `secrets` :** `secrets` utilise un générateur aléatoire cryptographiquement sûr, conçu pour résister aux attaques.
— **Génération du token :** `secrets.token_urlsafe(32)` produit :
  — 32 octets aléatoires,
  — encodés au format Base64 URL-safe,
  — ce qui donne un token long, imprévisible et compatible avec les URLs.
— **Utilité :** Ce type de token est utilisé dans :
  — systèmes de confirmation d'email,
  — liens de réinitialisation de mot de passe,
  — API sécurisées,
  — sessions temporaires.



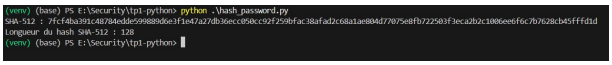
3.3 Hachage SHA-256 / SHA-512

```python
import hashlib

password = "admin123".encode()
sha512_hash = hashlib.sha512(password).hexdigest()
```

```
5 sha256_hash = hashlib.sha256(password).hexdigest()
```

Explication du code

- **Objectif :** Calculer l'empreinte cryptographique (hash) d'un mot de passe à l'aide de deux fonctions de hachage sécurisées : `SHA-256` et `SHA-512`.
- **Encodage du mot de passe :** Le texte `"admin123"` est converti en bytes grâce à `encode()`, car les fonctions de hachage travaillent uniquement sur des données binaires.
- **SHA-256 :**
  - produit un hash de 256 bits (64 caractères hexadécimaux),
  - largement utilisé pour l'intégrité des données et dans les blockchains,
  - très résistant aux collisions.
- **SHA-512 :**
  - version plus longue : 512 bits (128 caractères hexadécimaux),
  - encore plus robuste contre les attaques par force brute.
- **Méthode `hexdigest()` :** Transforme le résultat binaire en une chaîne hexadécimale lisible.
- **Remarque importante :** Le hachage est **irréversible**. Il ne permet pas de retrouver le mot de passe original. Pour un stockage sécurisé réel, on utilise un **hachage salé** (salt + hash), comme PBKDF2 ou bcrypt.



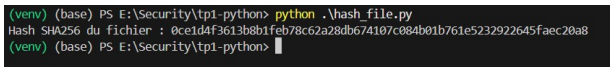
3.4 Hachage d'un fichier

```
1 import hashlib
2
3 with open("image.jpg", "rb") as f:
4     data = f.read()
5
6 h = hashlib.sha256(data).hexdigest()
7 print("Hash␣SHA256␣du␣fichier␣:", h)
```

Explication du code

- **Objectif :** Calculer l'empreinte SHA-256 d'un fichier (ici `image.jpg`) afin d'en vérifier l'intégrité.
- **Ouverture du fichier :** Le fichier est ouvert en mode binaire (`"rb"`), indispensable pour lire correctement des images ou tout autre contenu non textuel.
- **Lecture des données :** La totalité du fichier est chargée en mémoire dans la variable `data`.
- **Calcul du hash :** La fonction `hashlib.sha256()` reçoit les données et génère un condensat cryptographique de 256 bits, unique au contenu du fichier.

— **Affichage :** `hexdigest()` convertit le résultat binaire en une chaîne hexadécimale lisible (64 caractères).
— **Utilité :** Ce hachage permet de :
  — vérifier si un fichier a été modifié,
  — détecter une corruption ou une altération volontaire,
  — comparer deux fichiers de manière fiable (intégrité).



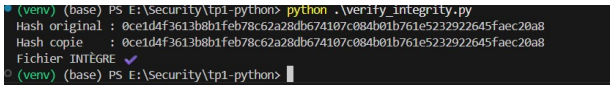
3.5 Vérification d'intégrité

```python
import hashlib

def hash_file(path):
    h = hashlib.sha256()
    with open(path, "rb") as f:
        h.update(f.read())
    return h.hexdigest()

original = hash_file("original.jpg")
copy = hash_file("copie.jpg")

if original == copy:
    print("Fichier␣INT GRE␣      ")
else:
    print("Fichier␣MODIFI ␣   ")
```

Explication du code

— **Objectif :** Vérifier si deux fichiers (un original et une copie) sont identiques en comparant leurs empreintes SHA-256.
— **Fonction `hash_file(path)` :**
  — crée un objet de hachage SHA-256,
  — ouvre le fichier en mode binaire (`"rb"`),
  — lit son contenu complet,
  — met à jour le hachage avec les données lues,
  — renvoie l'empreinte sous forme hexadécimale (64 caractères).
— **Calcul des deux empreintes :**
  — `original` contient le hash de `original.jpg`,
  — `copy` contient le hash de `copie.jpg`.
— **Comparaison :** Si les deux empreintes sont strictement égales :
  — le contenu des deux fichiers est **identique**, → affichage : `Fichier INTÈGRE`
  Sinon :
  — le fichier a été modifié, altéré ou corrompu, → affichage : `Fichier MODIFIÉ`
— **Pourquoi c'est fiable ?** `SHA-256` est conçu pour que :

— deux fichiers différents aient des empreintes totalement différentes,
— il soit impossible de retrouver le fichier original à partir du hash.
Ainsi, la comparaison des empreintes est un moyen sûr de vérifier l'intégrité d'un fichier.



Conclusion

Ce TP nous a permis de maîtriser et de manipuler concrètement les principaux outils cryptographiques utilisés en sécurité informatique. À travers l'ensemble des exercices réalisés, nous avons pu explorer :

- le hachage cryptographique (MD5, SHA-256, SHA-512) et le contrôle d'intégrité des données,
- le chiffrement symétrique avec DES et AES (incluant padding, IV, CBC),
- la dérivation de clés sécurisées via PBKDF2 et PBKDF2HMAC,
- le chiffrement et la signature RSA (génération de clés, signature, vérification, OAEP),
- l'utilisation du module `cryptography` : Fernet, AES bas niveau, PBKDF2,
- la génération de mots de passe et de tokens sécurisés avec `secrets`,
- la vérification d'intégrité des fichiers par comparaison d'empreintes.

L'ensemble de ces manipulations nous a permis de consolider notre compréhension pratique des mécanismes essentiels garantissant la confidentialité, l'intégrité et l'authenticité des données. Ces compétences constituent une base solide pour des travaux avancés en cybersécurité.

Enfin, afin de rendre ce travail reproductible et accessible, un dépôt GitHub dédié a été créé. Il contient **tous les scripts Python, les fichiers générés, les clés RSA et les exemples utilisés au cours du TP**. Le dépôt est disponible à l'adresse suivante :

`https://github.com/Jalil03/security-tp01-python`

Ce repository permet de retrouver facilement l'intégralité du code, de refaire les tests, et de poursuivre les expérimentations cryptographiques de manière autonome.